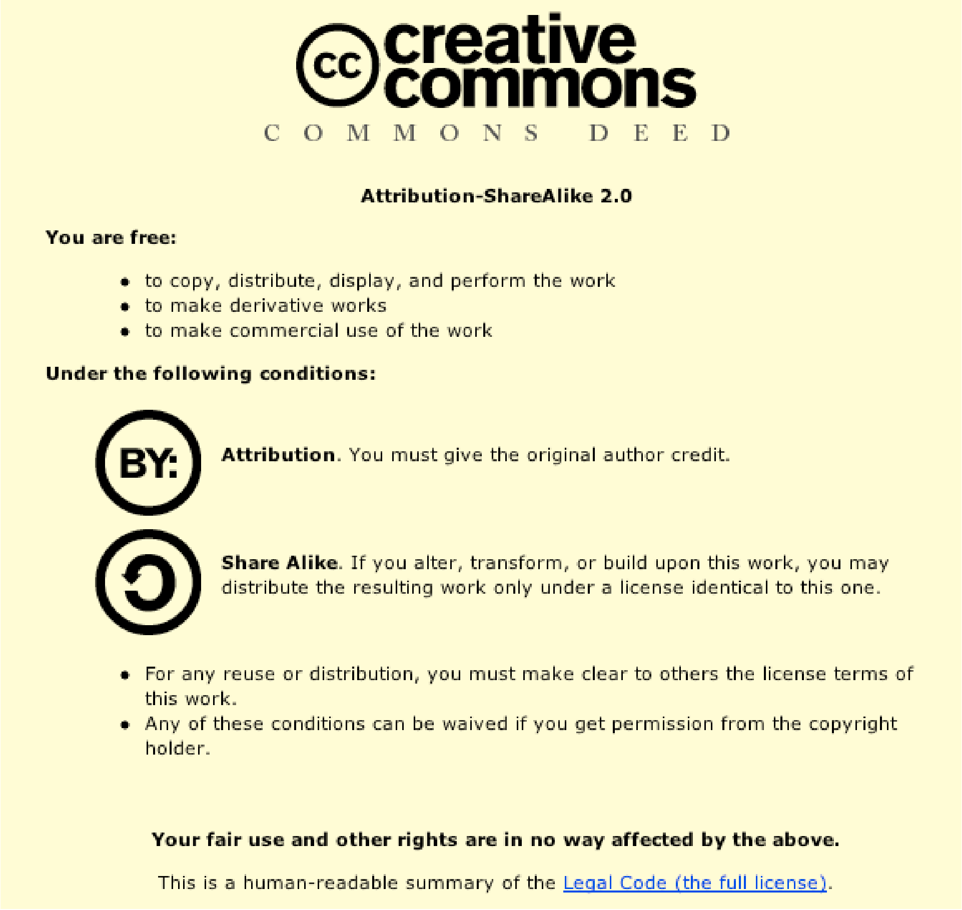
Some Design Points

Stéphane Ducasse
Stephane.Ducasse@univ-savoie.fr
http://www.listic.univ-savoie.fr/~ducasse/

Stéphane Ducasse --- 2005

License: CC-Attribution-ShareAlike 2.0

http://creativecommons.org/licenses/by-sa/2.0/



COMMONS DEED

Attribution-ShareAlike 2.0

You are free:

- to copy, distribute, display, and perform the work
- to make derivative works
- to make commercial use of the work

Under the following conditions:

Attribution. You must give the original author credit.

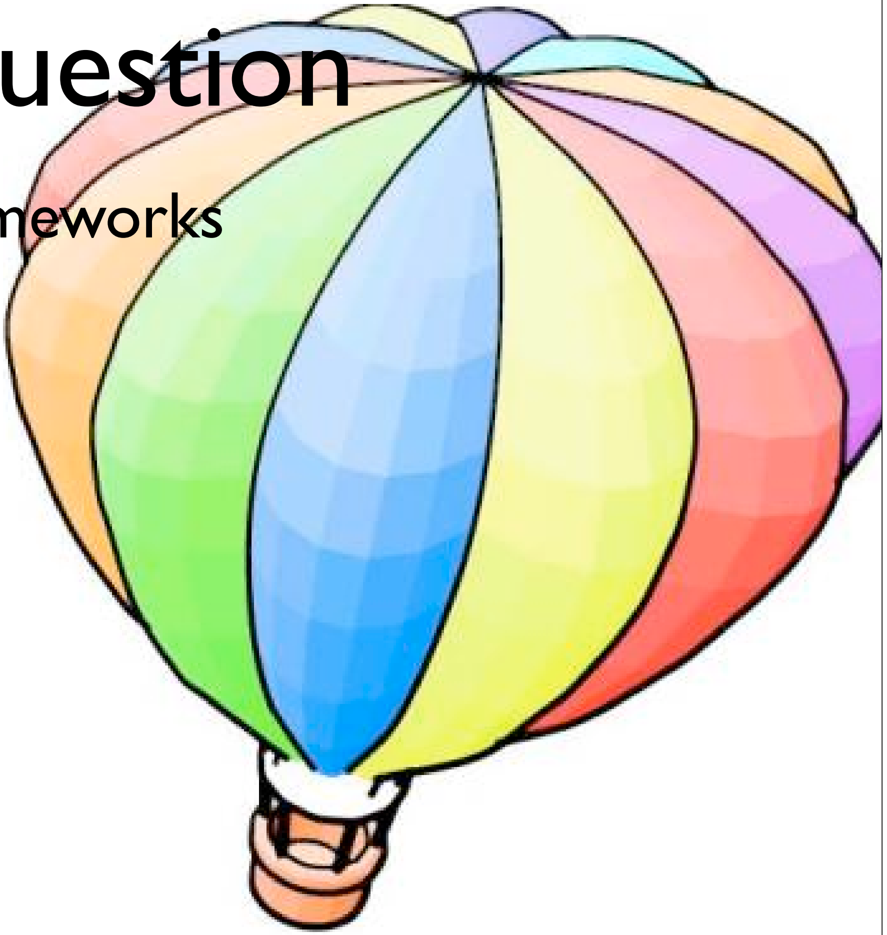
Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.

- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.

This is a human-readable summary of the Legal Code (the full license).

The Design in Question



- The Basic Idea behind Frameworks
- Subclassing vs SubTyping
- Coupling
- Design Heuristics
- Design Symptoms

Frameworks

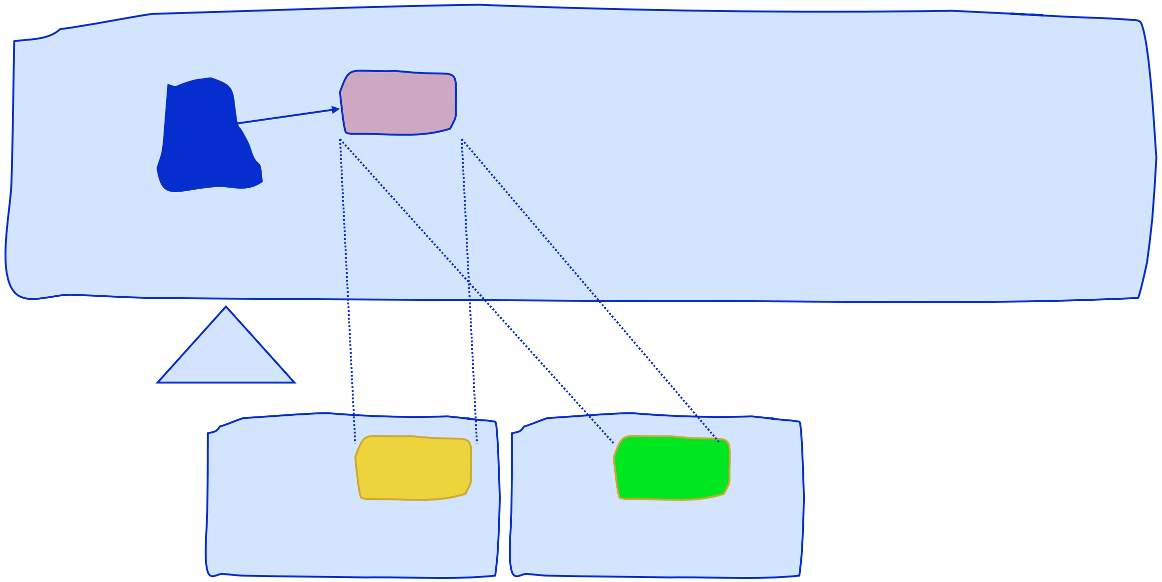
- What is it?
- Principles
- vs. Libraries

Inheritance as Parameterization

- Subclass customizes hook methods by implementing (abstract) operations in the context of template method
- Any method acts as a parameter of the context
- Methods are unit of reuse
- Abstract class -- one that must be customized before it can be used

Methods are Unit of Reuse

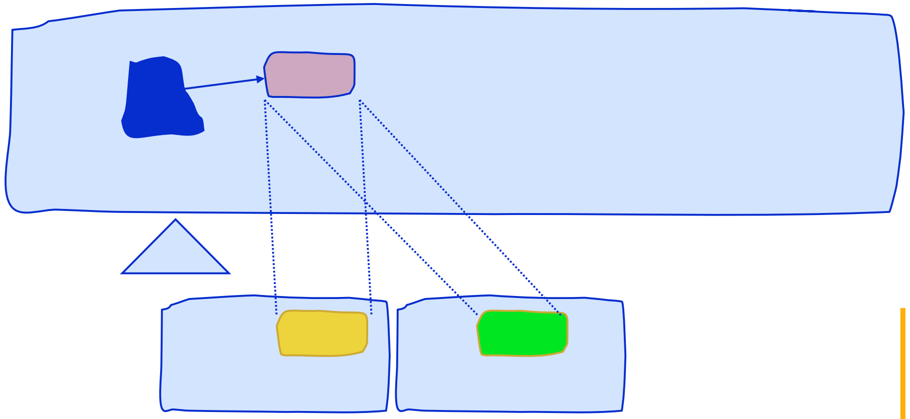
self sends are plans for reuse



🟪 can be abstract or not

Frameworks vs. Libraries

- Libraries
  - You call them
  - Callback to extend them
- Framework
  - **Hollywood principle:** *Don't call me I will call you*
  - ***GreyHound principle:*** *Let's drive*

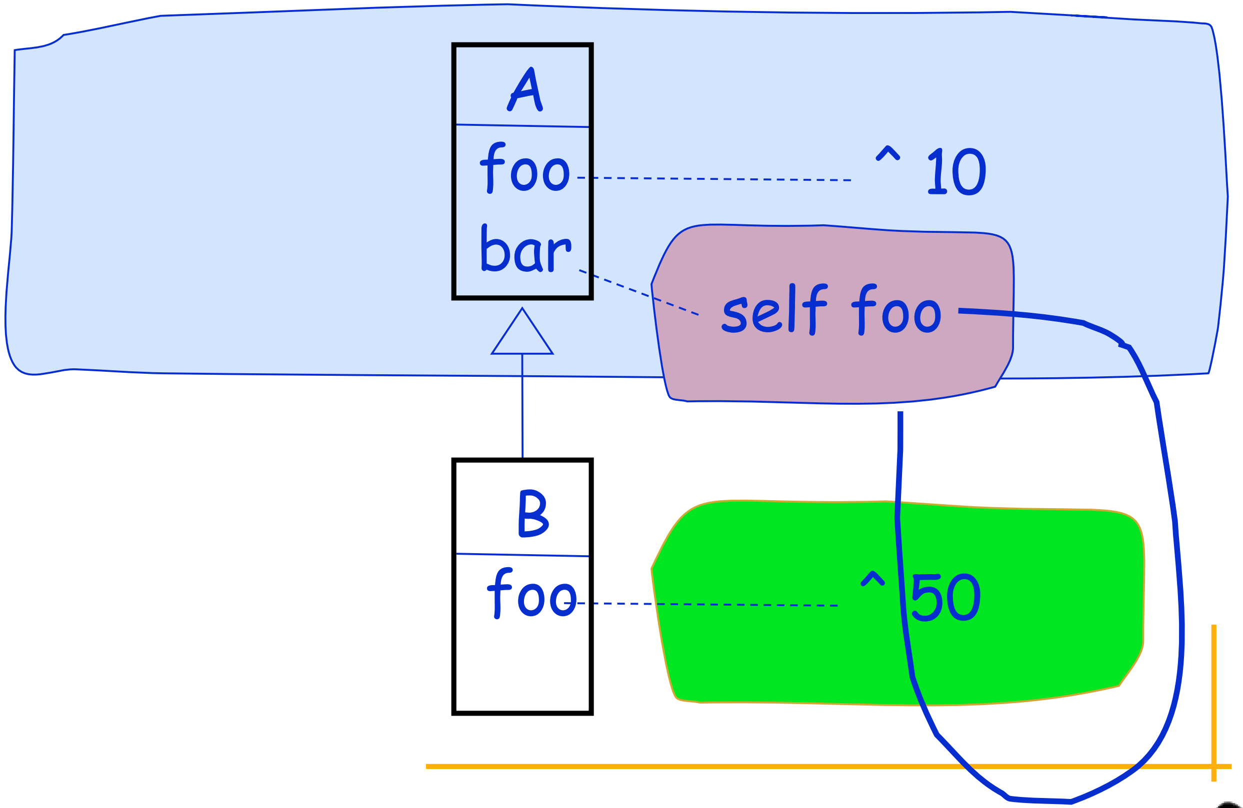


Library vs. Framework

Classes instantiated by the client	Framework instantiated classes, extended by inheritance
Clients invoke library functions	Framework calls the client functions
No predefined flow, predefined interaction, default behavior	Predefined flow, interaction and default behavior

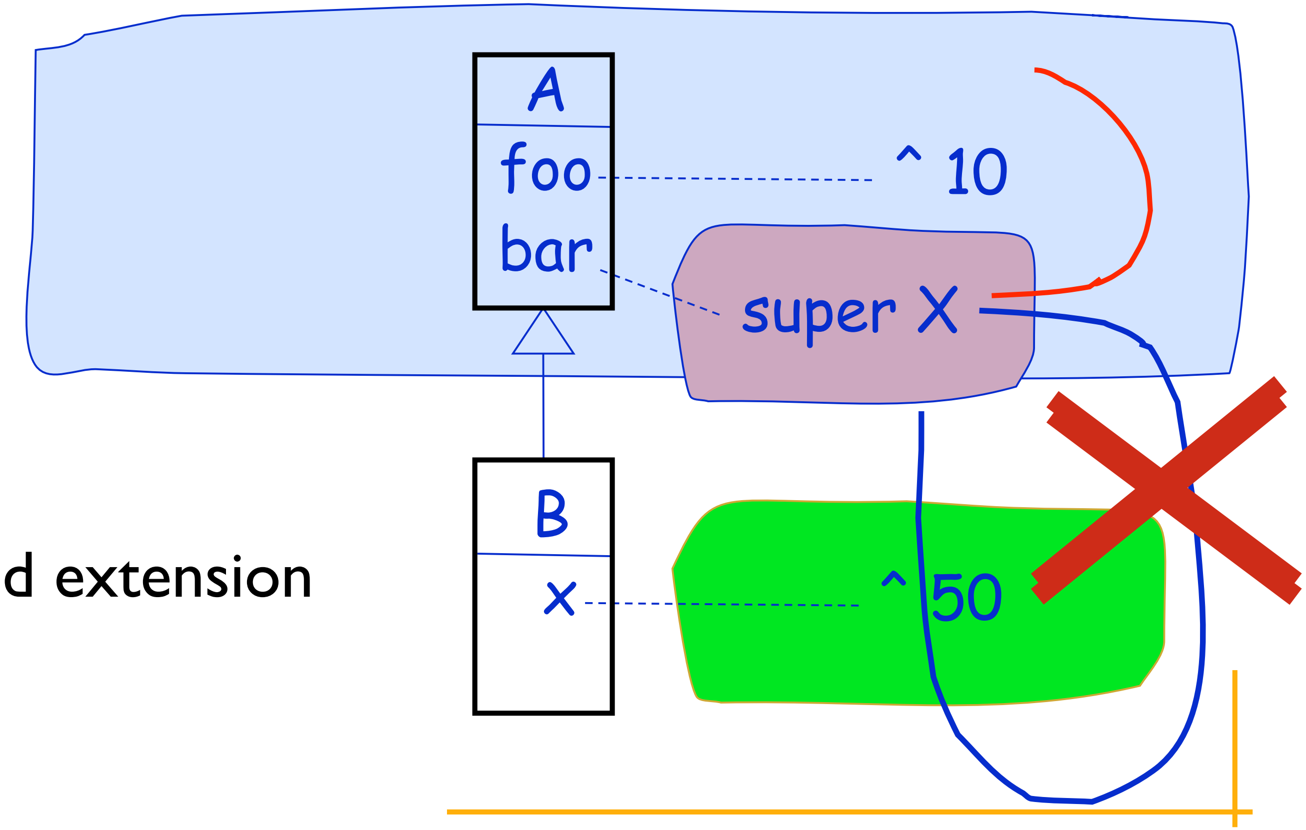
You remember self…

- self is dynamic
- self acts as a hook



You remember super…

- super is static



- super forbid extension

Frameworks

- A set of collaborating classes that define a context and are reusable by extension in different applications
- A framework is a reusable design expressed as a set of abstract classes and the way their instances collaborate. By definition, a framework is an object-oriented design. It doesn't have to be implemented in an object-oriented language, though it usually is. Large-scale reuse of object-oriented libraries requires frameworks. The framework provides a context for the components in the library to be reused. [Johnson]
- A framework often defines the architecture of a set of applications

On Frameworks...

- Frameworks design
  - Need at least 3 applications to support the generalization
  - http://st-www.cs.uiuc.edu/users/droberts/evolve.html
- Smile if somebody tell that they start implementing a framework
- Framework often rely on whitebox abstractions: ie extended by inheritance
- Others are blackboxes framework: ie extended by composition
- A framework can use design patterns

SubTyping vs. Subclassing

How to Implement a Stack?

By subclassing OrderedCollection...

```
Stack>>pop
   ^ self removeLast
Stack>>push: anObject
   self addFirst: anObject
Stack>>top
   ^ self first
```

Stack>>size, Stack>>includes:
are free, inherited from

BUT BUT BUT!!!

- What do we do with all the rest of the interface of OrderedCollection?

- a Stack IS NOT an OrderedCollection!
- We cannot substitute an OrderedCollection by a Stack

- Some messages do not make sense on Stack
    - Stack new addLast: anObject
- Stack new last

- So we have to block a lot of methods...

Consequences...

```
Stack>>removeLast
    self shouldNotImplement

Stack>>pop
   ^ super removeLast
```

The Problem

- There is not a clean simple relationship between Stack and OrderedCollection

- Stack interface is not an extension or subset of OrderedCollection interface

- Compare with CountingStack a subclass of Stack
- CountingStack is an extension

Another Approach

By defining the class Stack that uses OrderedCollection

```
Object subclass: Stack
      iv: elements

Stack>>push: anElement
    elements addFirst: anElement
Stack>>pop
    element isEmpty ifFalse: [^ self removeFirst]
```

Inheritance and Polymorphism

- Polymorphism works best with standard interfaces
- Inheritance creates families of classes with similar interfaces
- Abstract class describes standard interfaces
- Inheritance helps software reuse by making polymorphism easier

Specification Inheritance

- Subtyping
- Reuse of specification
  - A program that works with Numbers will work with Fractions.
  - A program that works with Collections will work with Arrays.

- A class is an abstract data type (Data + operations to manipulate it)

Inheritance for Code Reuse

- Subclassing
- Dictionary is a subclass of Set
- Semaphore is a subclass of LinkedList
- No relationship between the interfaces of the classes

- Subclass reuses code from superclass, but has a different specification. It cannot be used everywhere its superclass is used. Usually overrides a lot of code.

- ShouldNotImplement use is a bad smell…

Inheritance for Code Reuse

- Inheritance for code reuse is good for
- rapid prototyping
  - getting application done quickly.
- Bad for:
  - easy to understand systems
  - reusable software
  - application with long life-time.

Subtyping Essence

- You reuse specification
  - You should be able to substitute an instance by one of its subclasses (more or less)
  - There is a relationship between the interfaces of the class and its superclass

How to Choose?

- Favor subtyping
- When you are in a hurry, do what seems easiest.
- Clean up later, make sure classes use "is-a" relationship, not just "is-implemented-like".
- Is-a is a design decision, the compiler only enforces is-implemented-like!!!

Quizz

- Circle subclass of Point?
- Poem subclass of OrderedCollection?

Class Design

Behavior Up and State Down

- Define classes by behavior, not state
- Implement behavior with abstract state: if you need state do it indirectly via messages.
- Do not reference the state variables directly
- Identify message layers: implement class's behavior through a small set of kernel method

Example

```
Collection>>removeAll: aCollection
        aCollection do: [:each | self remove: each]
        ^ aCollection

Collection>>remove: oldObject
        self remove: oldObject ifAbsent: [self notFoundError]

Collection>>remove: anObject ifAbsent:
anExceptionBlock
        self subclassResponsibility
```

Behavior-Defined Class

When creating a new class, define its public protocol and specify its behavior without regard to data structure (such as instance variables, class variables, and so on).

For example:

Rectangle

Protocol:

- area
- intersects:
- contains:
- perimeter

Implement Behavior with Abstract State

- If state is needed to complete the implementation
- Identify the state by defining a message that returns that state instead of defining a variable.

For example, use

```
Circle>>area
        ^self radius squared * self pi
```

not

```
Circle>>area
        ^radius squared * pi.
```

Identify Message Layers

- How can methods be factored to make the class both efficient and simple to subclass?
- Identify a small subset of the abstract state and behavior methods which all other methods can rely on as kernel methods.

```
Circle>>radius
Circle>>pi
Circle>>center
Circle>>diameter
        ^self radius * 2
Circle>>area
        ^self radius squared * self pi
```

Good Coding Practices

- Good Coding Practices promote good design
- Encapsulation
- Level of decomposition
- Factoring constants

The Object Manifesto

- Be lazy and be private
- Never do the job that you can delegate to another one
- Never let someone else plays with your private data

The Programmer Manifesto

- Say something only once
- Don’t ask, tell!

Tell, Don’t Ask!

```
MyWindow>>displayObject: aGrObject
    aGrObject displayOn: self
```

- And not:

```
MyWindow>>displayObject: aGrObject

    aGrObject isSquare ifTrue: […]
    aGrObject isCircle ifTrue: […]
    …
```

Good Signs of OO Thinking

- Short methods
- No dense methods
- No super-intelligent objects
- No manager objects
- Objects with clear responsibilities
  - State the purpose of the class in one sentence
- Not too many instance variables

Composed Methods

- How do you divide a program into methods?
  - Messages take time
  - Flow of control is difficult with small methods
- But:
  - Reading is improved
  - Performance tuning is simpler (Cache...)
  - Easier to maintain / inheritance impact

Composed Methods

- Divide your program into methods that perform one identifiable task. Keep all of the operations in a method at the same level of abstraction.

- Controller>>controlActivity

```
self controlInitialize.
self controlLoop.
self controlTerminate
```

Do you See the Problem?

```
initializeToStandAlone

	super initializeToStandAlone.
	self borderWidth: 2.
	self borderColor: Color black.
	self color: Color blue muchLighter.
	self extent: self class defaultTileSize * (self columnNumber @ self rowNumber).
	self initializeBots.
	self running.
	area := Matrix rows: self rowNumber columns: self columnNumber.
	area indicesDo: [:row :column | area at: row at: column
						put: OrderedCollection new].
	self fillWorldWithGround.
	self firstArea.
	self installCurrentArea
```

Do you See the Problem?

```
initializeToStandAlone

    super initializeToStandAlone.
    self initializeBoardLayout.
    self initializeBots.
    self running.
    self initializeArea.
    self fillWorldWithGround.
    self firstArea.
    self installCurrentArea
```

With code reuse…

```
initializeArea

	area := self matrixClass
			rows: self rowNumber
			columns: self columnNumber.
	area indicesDo: [:row :column | area
					at: row
					at: column
					put: OrderedCollection new]
```

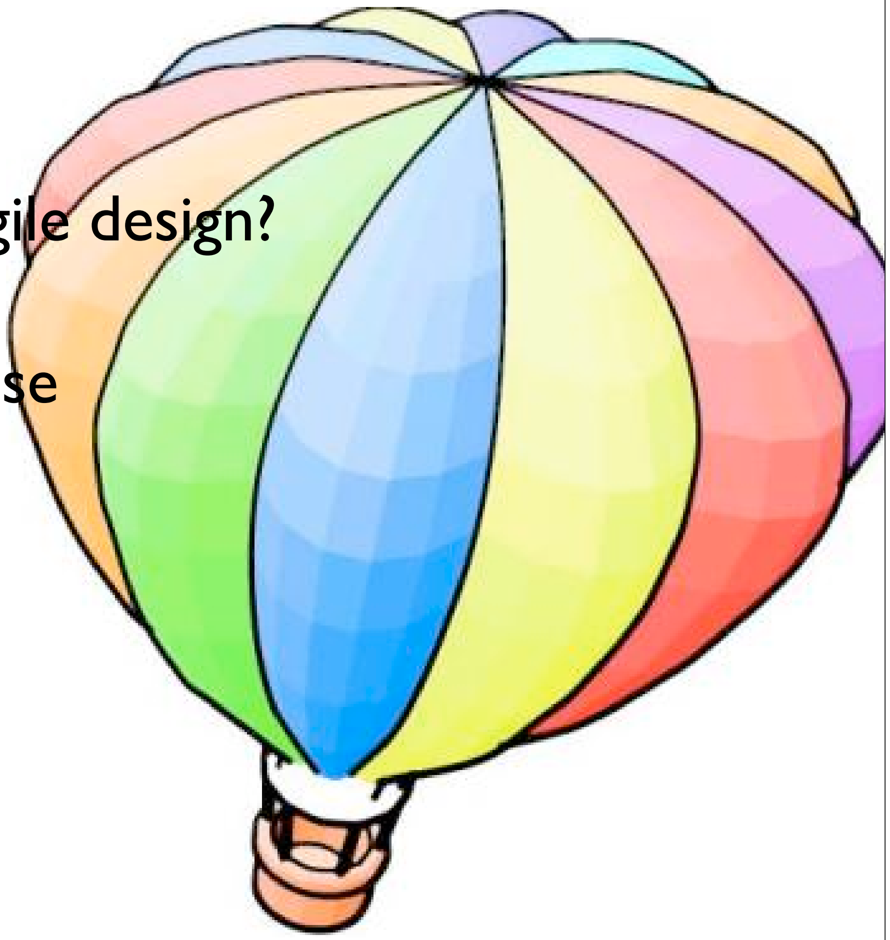
initializeArea can be invoke ***several*** times

About Methods

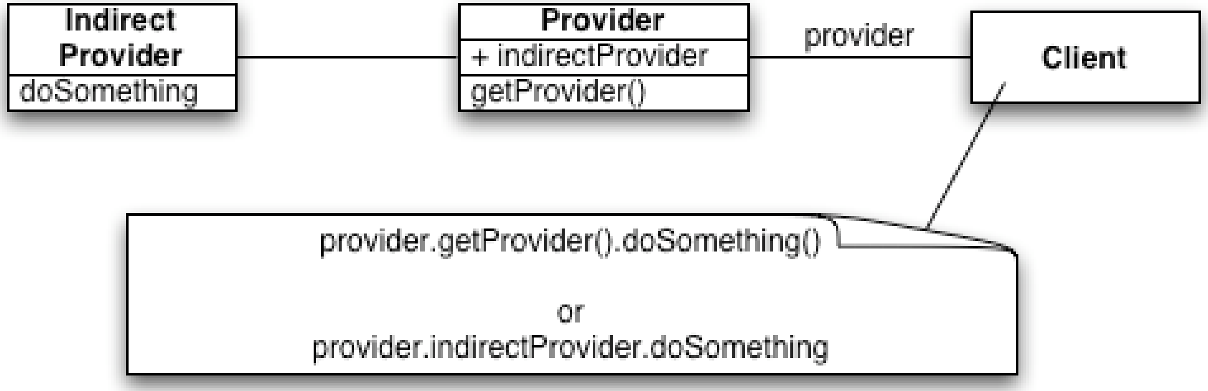
- Avoid long methods
- A method: one task
- Avoid duplicated code
- Reuse Logic

About Coupling

- Why coupled classes is fragile design?
- Law of Demeter
- Thoughts about accessor use



The Core of the Problem



Indirect Provider
doSomething

Provider
+ indirectProvider
getProvider()

provider

Client

provider.getProvider().doSomething()

or
provider.indirectProvider.doSomething

The Law of Demeter

- You should only send messages to:
  - an argument passed to you
  - an object you create
  - self, super
  - your class
- Avoid global variables
- Avoid objects returned from message sends other than self

Correct Messages

```
someMethod: aParameter
	self foo.
	super someMethod: aParameter.
	self class foo.
	self instVarOne foo.
	instVarOne foo.
	self classVarOne foo.
	classVarOne foo.
	aParameter foo.
	thing := Thing new.
	thing foo
```

Law of Demeter by Example

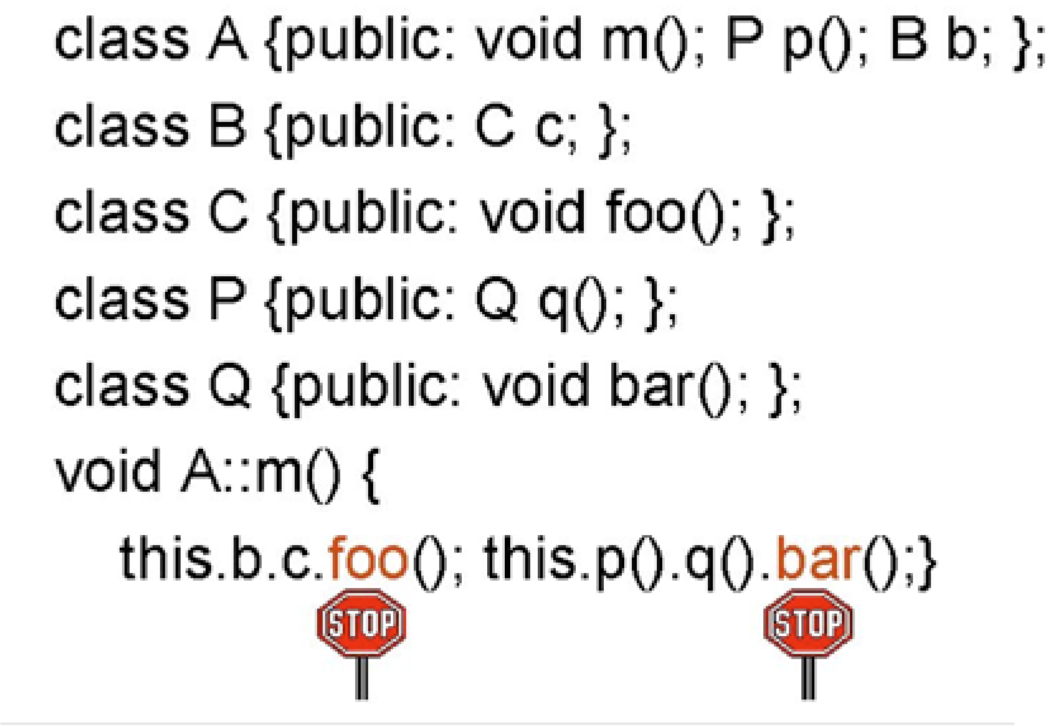
```
NodeManager>>declareNewNode: aNode
    |nodeDescription|
        (aNode isValid)                    "Ok passed as an
        argument to me"
            ifTrue: [ aNode certified].
    nodeDescription := NodeDescription for: aNode.
    nodeDescription localTime.                         "I created it"
        self addNodeDescription: nodeDescription.

        "I can talk to myself"
    nodeDescription data              "Wrong I should not know"
        at: self creatorKey               "that data is a dictionary"
        put: self creator
```

In other words

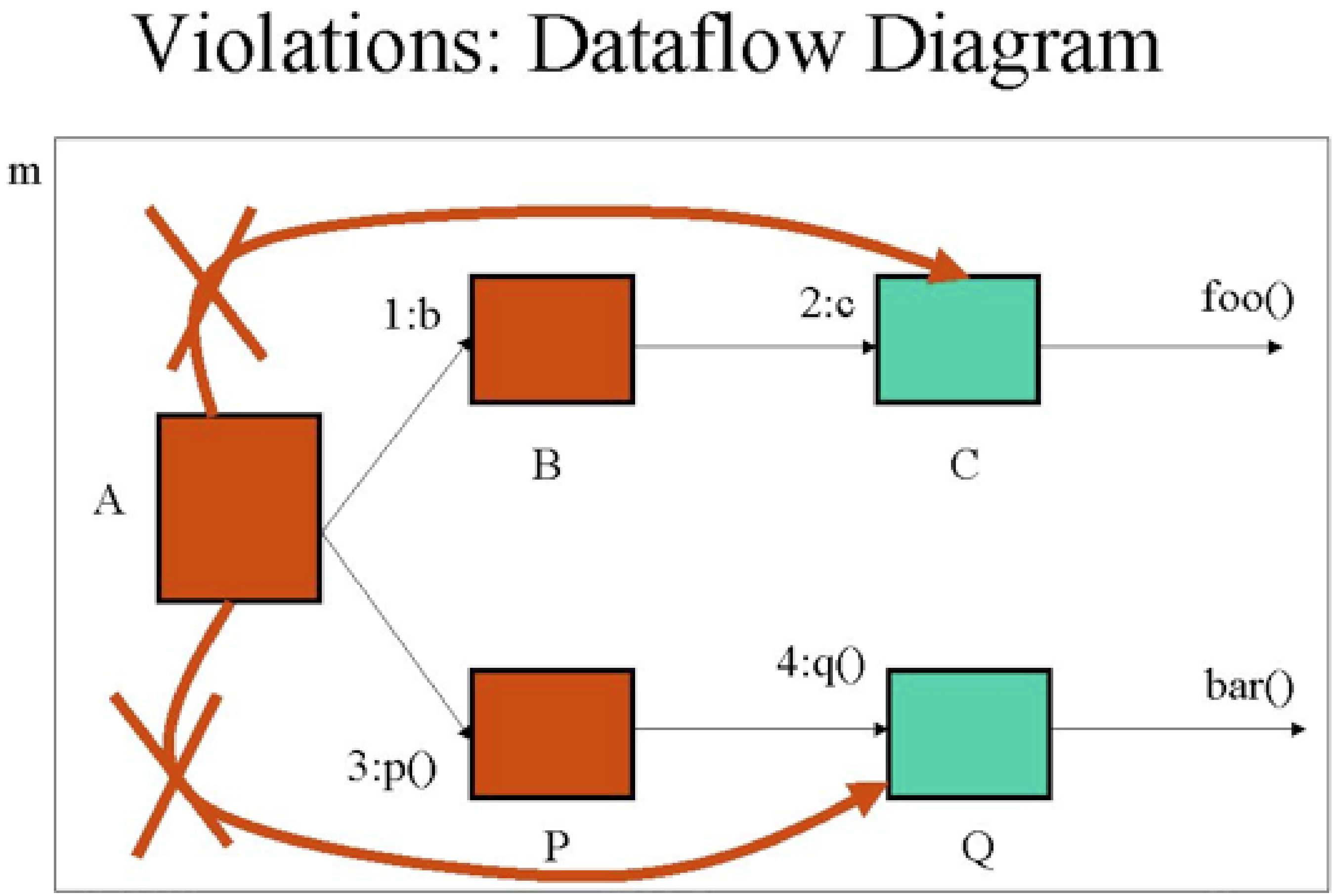
- Only talk to your immediate friends.
- In other words:
  - You can play with yourself. (this.method())
  - You can play with your own toys (but you can't take them apart). (field.method(), field.getX())
  - You can play with toys that were given to you. (arg.method())
  - And you can play with toys you've made yourself. (A a = new A(); a.method())

Halt!



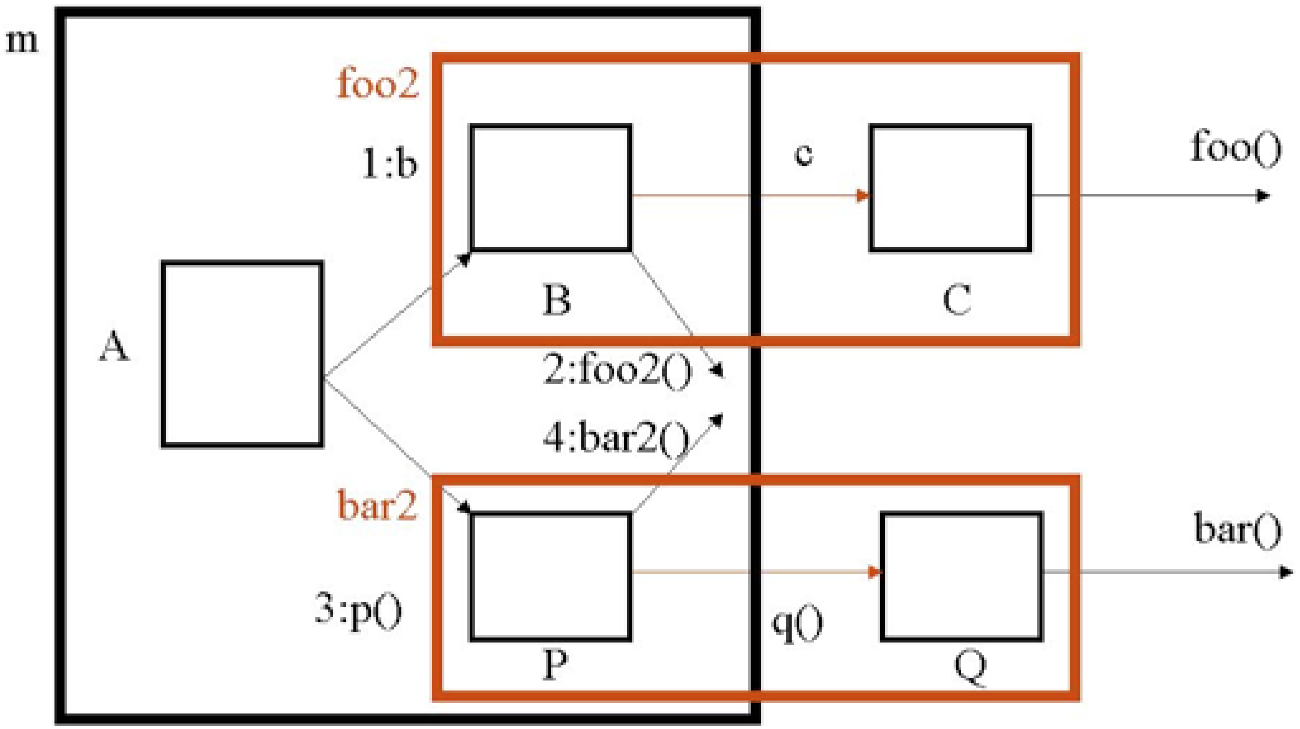
```
class A {public: void m(); P p(); B b; };
class B {public: C c; };
class C {public: void foo(); };
class P {public: Q q(); };
class Q {public: void bar(); };
void A::m() {
    this.b.c.foo(); this.p().q().bar();}
```

To not skip your intermediate

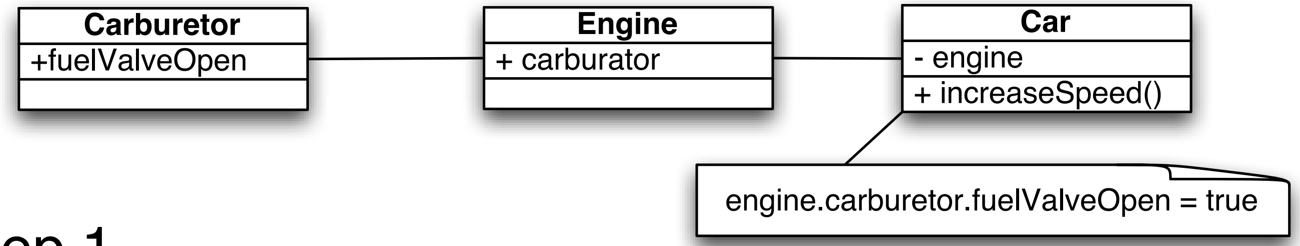


Solution

OO Following of LoD



Transformation



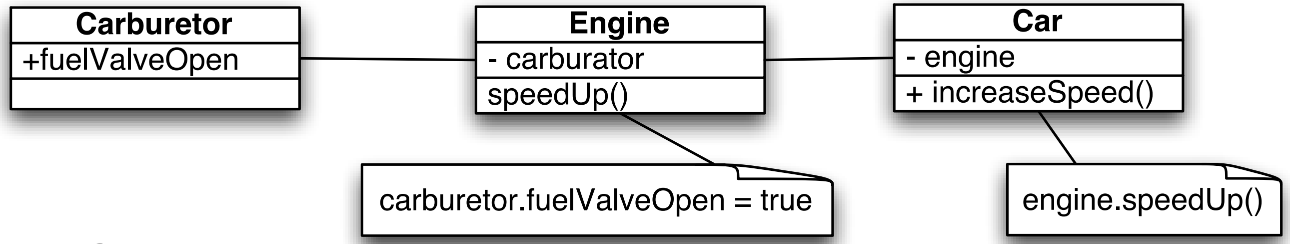
Carburetor
+fuelValveOpen

Engine
+ carburator

Car
- engine
+ increaseSpeed()

engine.carburetor.fuelValveOpen = true

Step 1



Carburetor
+fuelValveOpen

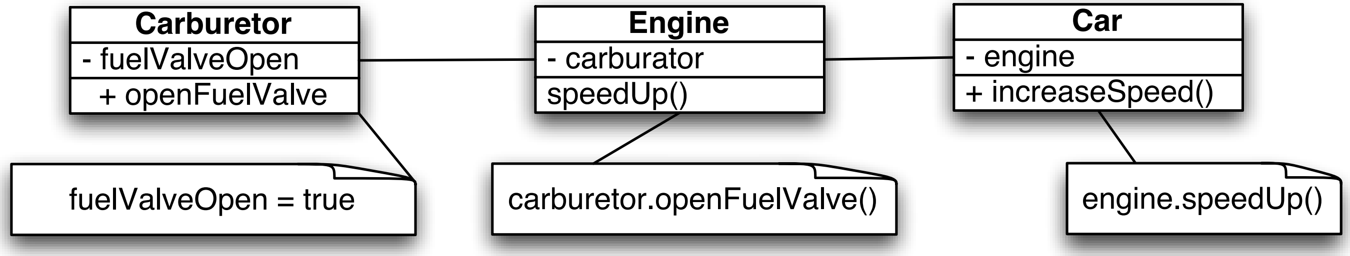
Engine
- carburator
speedUp()

Car
- engine
+ increaseSpeed()

carburetor.fuelValveOpen = true

engine.speedUp()

Step 2



Carburetor
- fuelValveOpen
+ openFuelValve

Engine
- carburator
speedUp()

Car
- engine
+ increaseSpeed()

fuelValveOpen = true

carburetor.openFuelValve()

engine.speedUp()

Law of Demeter’s Dark Side

```
Class A
	instVar: myCollection

A>>do: aBlock
		myCollection do: aBlock
A>>collect: aBlock
		^ myCollection collect: aBlock
A>>select: aBlock
		^ myCollection select: aBlock
A>>detect: aBlock
		^ myCollection detect: aBlock
A>>isEmpty
		^ myCollection isEmpty
```

....

About the Use of Accessors

Some schools say: "Access instance variables using methods"

But

- Be consistent inside a class, do not mix direct access and accessor use
- First think accessors as private methods that should not be invoked by clients
- Only when necessary put accessors in accessing protocol

Example

```
Scheduler>>initialize
          self tasks: OrderedCollection new.

      Scheduler>>tasks
          ^ tasks
```

But now everybody can tweak the tasks!

Accessors

Accessors are good for lazy initialization

```
Scheduler>>tasks
    tasks isNil ifTrue: [task := ...].
    ^ tasks
```

BUT accessors methods should be PRIVATE by default at least at the beginning

Accessors open Encapsulation

The fact that accessors are methods doesn’t support a good data encapsulation.

You could be tempted to write in a client:

```
ScheduledView>>addTaskButton
	...
	model tasks add: newTask
```

What’s happen if we change the representation of tasks?

Tasks

If tasks is now an array it will break

Take care about the coupling between your objects and provide a good interface!

```
   Schedule>>addTask: aTask
            tasks add: aTask

  ScheduledView>>addTaskButton
              ...
              model addTask: newTask
```

About Copy Accessor

Should I copy the structure?

```
Scheduler>>tasks
    ^ tasks copy
```

But then the clients can get confused...

```
Scheduler uniqueInstance tasks removeFirst
```

and nothing happens!

Use intention revealing names

Better

```
Scheduler>>taskCopy
   "returns a copy of the pending tasks"

    ^ task copy
```

Provide a Complete Interface

```
Workstation>>accept: aPacket
            aPacket addressee = self name
                ...
```

It is the responsibility of an object to propose a complete interface that protects itself from client intrusion.

Shift the responsibility to the Packet object

```
Packet>>isAddressedTo: aNode
       ^ addressee = aNode name
Workstation>>accept: aPacket
       (aPacket isAddressedTo: self)
```

Open-Close

- Software entities (classes, modules, functions, etc.) should be ***open*** for extension, but ***closed*** for modification.

The open-closed principle

- Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.

- Existing code should not be changed – new features can be added using inheritance or composition.

One kind of application

```
enum ShapeType {circle,
   square};
struct Shape {
   ShapeType _type;
};
struct Circle {
   ShapeType _type;
   double _radius;
   Point _center;
};
```

```
struct Square {
   ShapeType _type;
   double _side;
   Point _topLeft;
};
void DrawSquare
   (struct Square*)
void DrawCircle
   (struct Circle*);
```

Example (II)

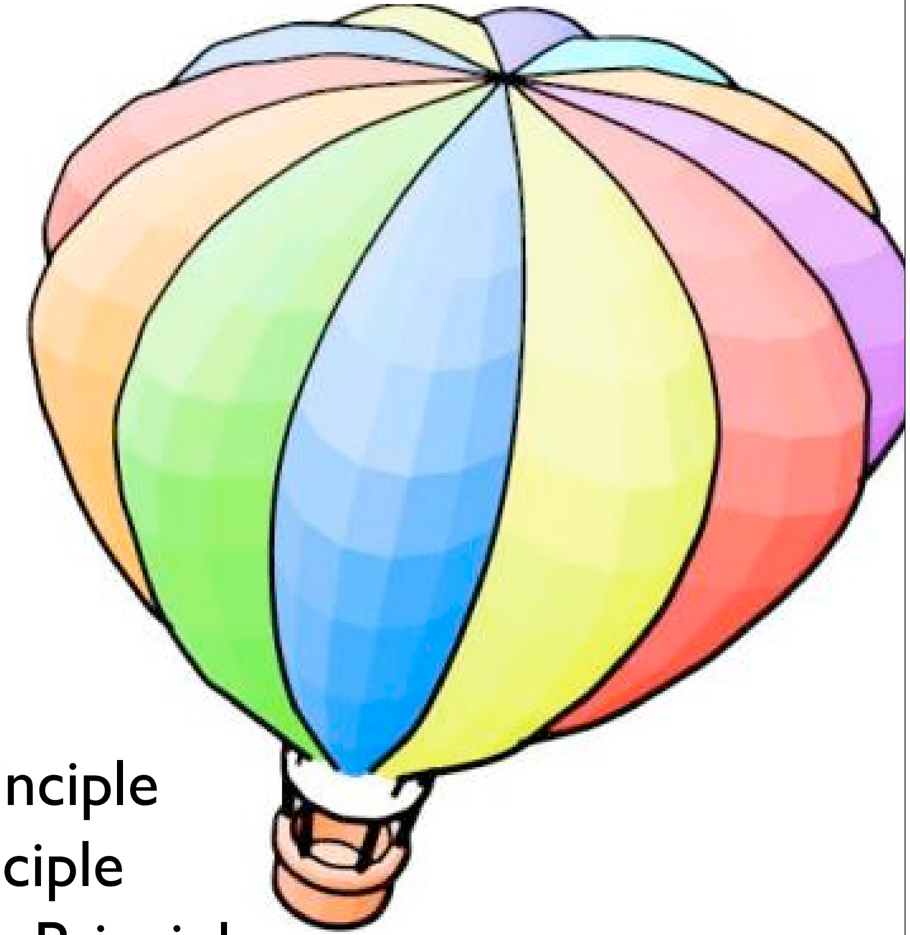
```
void DrawAllShapes(struct Shape* list[], int n) {
    int i;
    for (i=0; i<n; i++) {
        struct Shape* s = list[i];
        switch (s->_type) {
        case square: DrawSquare((struct Square*)s); break;
        case circle: DrawCircle((struct Circle*)s); break;
        }
    }
}
```

Adding a new shape requires adding new code to this method.

Correct Form

```
class Shape {
    public: virtual void Draw() const = 0;
};
class Square : public Shape {
    public: virtual void Draw() const;
};
class Circle : public Shape {
    public: virtual void Draw() const;
};
void DrawAllShapes(Set<Shape*>& list) {
    for (Iterator<Shape*>i(list); i; i++)
        (*i)->Draw();
```

Some Principles



- Dependency Inversion Principle
- Interface Segregation Principle
- The Acyclic Dependencies Principle

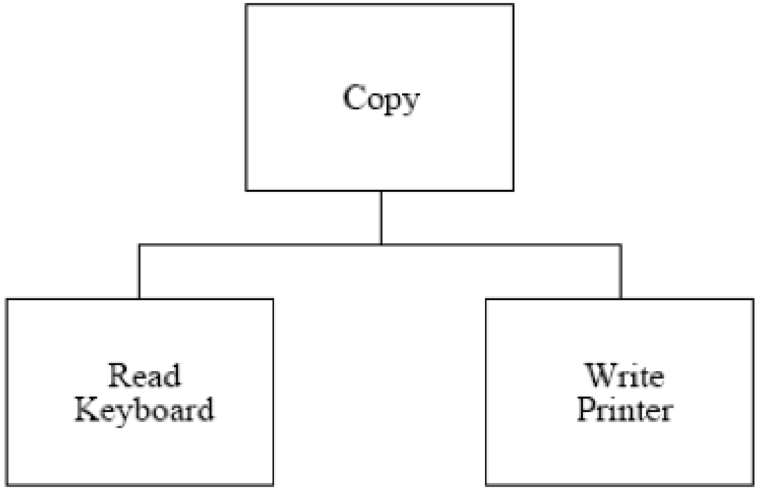
Dependency Inversion Principle

- High level modules should not depend upon low level modules. Both should depend upon abstractions.

- Abstractions should not depend upon details. Details should depend upon abstractions.

Example

```
void Copy() {
   int c;
   while ((c = ReadKeyboard()) != EOF)
     WritePrinter(c);
}
```



Cont...

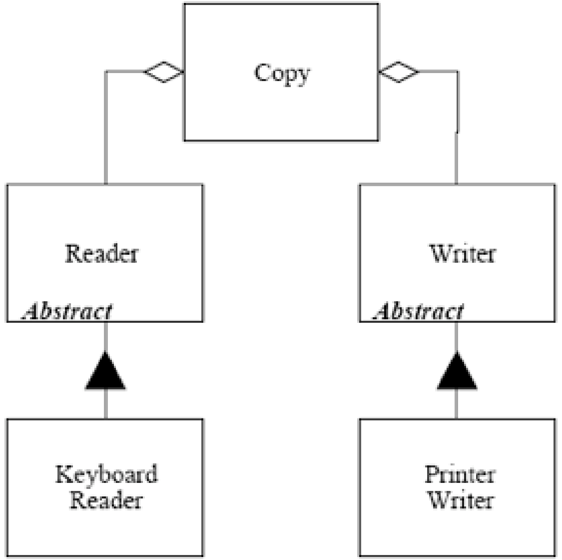
Now we have a second writing device – disk

```
enum OutputDevice {printer, disk};

void Copy(outputDevice dev) {
  int c;
  while ((c = ReadKeyboard()) != EOF)
    if (dev == printer)
      WritePrinter(c);
    else
      WriteDisk(c);
}
```

Solution

```
class Reader {
  public:
  virtual int Read() = 0;
};
class Writer {
  public:
    virtual void Write(char)=0;
};
void Copy(Reader& r,
          Writer& w) {
  int c;
  while((c=r.Read()) != EOF)
    w.Write(c);
}
```



Some Principle

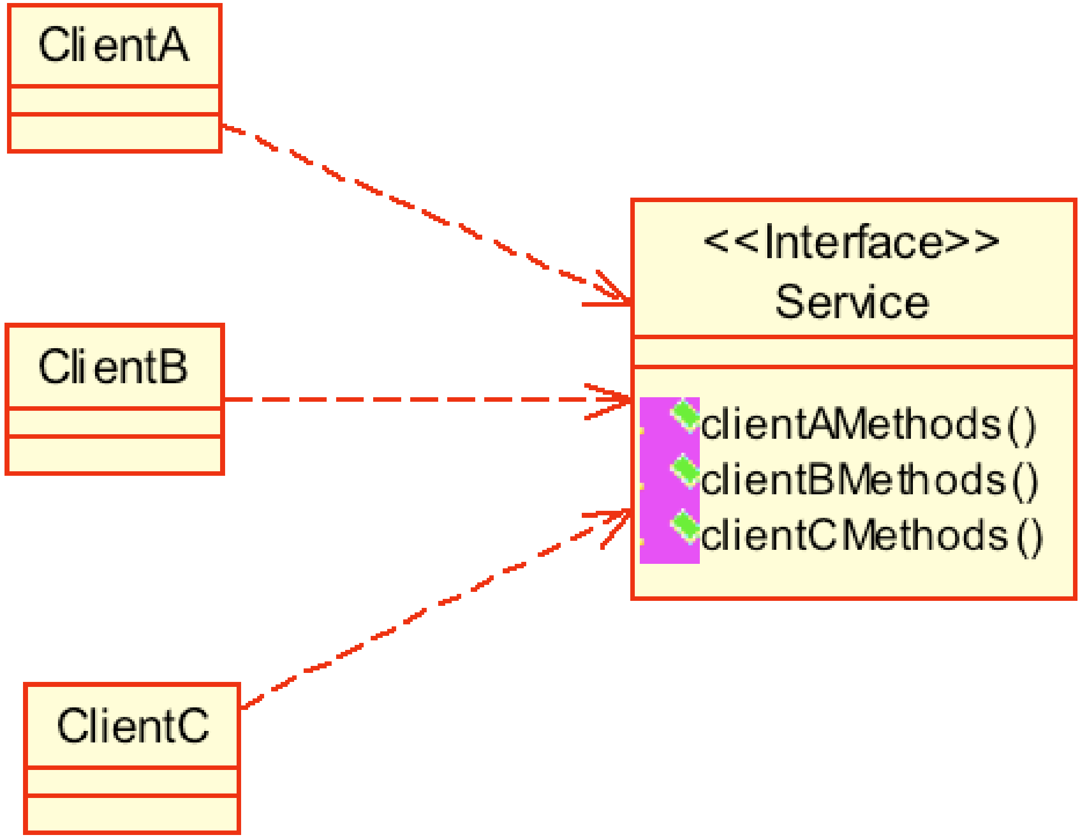
-
-
-
-
-
-
-
-

Interface Segregation Principle

- The dependency of one class to another one should depend on the smallest possible interface.

- Avoid “fat” interfaces

Examples


ClientA
ClientB
ClientC
<<Interface>>
Service
clientAMethods()
clientBMethods()
clientCMethods()

Solutions

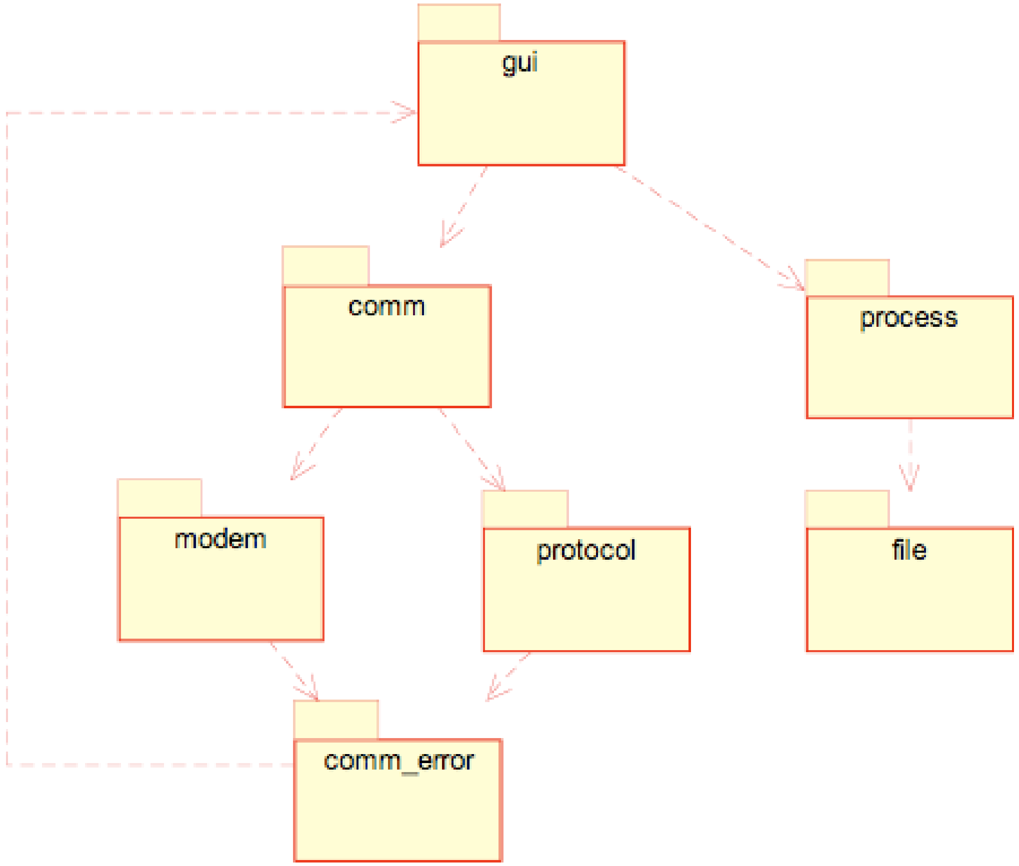
- One class one responsibility
- Composition?

- Design is not simple

The Acyclic Dependency Principle

- The dependency structure between packages must not contain cyclic dependencies.

Example...Ez



Solutions

- Layering?
- Separation of domain/applicatin/UI

Packages, Modules and other

- The Common Closure Principle
  - Classes within a released component should share common closure. That is, if one needs to be changed, they all are likely to need to be changed.

- The Common Reuse Principle
  - The classes in a package are reused together. If you reuse one of the classes in a package, you reuse them all.

Summary

Build your own taste

Analyze what you write and how?